Une stratégie de recherche basée sur la substituabilité

Mohammed Rezgui, Jean-Charles Régin, Arnaud Malapert

Une stratégie de recherche basée sur la substituabilité

Mohamed Rezgui **Jean-Charles Régin** **Arnaud Malapert**

Laboratoire i3S
Université de Nice-Sophia Antipolis

rezgui@i3s.unice.fr {Jean-Charles.REGIN, arnaud.malapert}@unice.fr

Résumé

Nous introduisons une nouvelle stratégie de recherche pour énumérer toutes les solutions d'un problème de satisfaction de contraintes. L'idée principale de cette stratégie consiste à énumérer des solutions génériques à partir desquelles toutes les solutions peuvent être efficacement calculées. Les solutions génériques contiennent des valeurs qui sont substituables à toutes les autres. Notre stratégie provoque l'apparition des valeurs substituables. Ainsi, à la différence d'une stratégie de recherche classique, notre méthode économise du temps en générant seulement quelques solutions génériques. Nous montrons expérimentalement que notre approche donne des résultats intéressants sur des problèmes ayant un grand nombre de solutions.

1 Introduction

Les problèmes de satisfaction de contraintes (CSP) constituent un cadre formel simple pour représenter et résoudre des problèmes en intelligence artificielle et en recherche opérationnelle. Dans cet article, nous nous focalisons sur le calcul de toutes les solutions d'un problème. Calculer toutes les solutions d'une instance permet par exemple de représenter certains sous-problèmes par une contrainte de table contenant comme combinaisons l'ensemble des solutions de ces sous-problèmes. Nous proposons une nouvelle stratégie de recherche basée sur la notion de substituabilité des valeurs que l'on nomme Substituability Based Search (SBS). **L'idée est de détecter les valeurs substituables et de forcer leur apparition afin d'obtenir des solutions génériques qui seront ensuite énumérées pour générer toutes les solutions. On espère ainsi parcourir moins de noeuds de l'arbre de recherche**.

Nous pouvons illustrer cela sur l'exemple suivant. On considère un problème Π impliquant une variable x et a et b, deux valeurs appartenant au domaine de x que l'on note $D(x)$. Supposons que (x, a) soit compatible avec toutes les valeurs de toutes les contraintes impliquant x alors a est substituable à toutes les autres valeurs de $D(x)$. Cela signifie qu'il ne peut y avoir de solution impliquant (x, b) qui ne soit pas une solution avec (x, a). Par ailleurs, si une affectation impliquant (x, a) est une solution alors il est facile de tester si cette affectation reste une solution quand on remplace (x, a) par (x, b). En effet, il suffit de tester la validité des contraintes. Ainsi, en cherchant toutes les solutions impliquant seulement (x, a), on peut énumérer toutes les solutions impliquant les autres valeurs de $D(x)$.
La SBS est basée sur cette idée. De plus, SBS va provoquer cette substituabilité en s'inspirant du principe de l'algorithme de Bron & Kerbosh [1]. On procède de la manière suivante. Soit (y_k, b_k), les couples (variable, valeur) non-supports de (x, a). En instanciant d'abord les couples (y_k, b_k) avant (x, a), on va provoquer la substituabilité de (x, a). En effet, au moment où (x, a) sera choisi, les couples (y_k, b_k) auront disparus et (x, a) sera compatible avec toutes les valeurs restantes des $D(y_k)$. Ces valeurs sont mises dans un ensemble associé à x que l'on nomme $TSIE_x$. Une solution générique est constituée d'ensembles de TSIE. L'énumération d'une solution se fait de la manière suivante. Lorsque toutes les variables sont instanciées, il faudra énumérer de manière efficace les solutions en vérifiant la compatibilité de chaque valeur a contenue dans $TSIE_x$ avec les autres valeurs contenues dans les autres ensembles $TSIE_{x'}$. Les ensembles $TSIE_x$ qui contiennent des valeurs compatibles en leur sein et aux autres valeurs de $TSIE_{x'}$ forment des séquences $Global\ Cut\ Seed$ (GCS), notion reprise de l'article [2]. Ainsi, nous proposons dans cet article un algo-

rithme permettant de calculer ces GCS qui seront ensuite énumérées pour générer toutes les solutions. L'article est organisé comme suit. Tout d'abord, nous rappelons certaines notions. Ensuite, nous exposons les algorithmes utilisés par SBS. Puis, nous présentons les résultats expérimentaux relatant les avantages et les inconvénients de SBS. Enfin, nous concluons sur les apports et les différentes possibilités qu'offre SBS.

2 Notations

Dans les sections suivantes, nous utiliserons les notations suivantes. Soit $X = [x_1, x_2, ..., x_n]$, la liste des variables d'un CSP binaire Π et $D(x_i)$, le domaine associé à une variable x_i. x_i et x_j sont deux variables distinctes avec $i < j$. Une valeur a_i représente une valeur de $D(x_i)$. On note (x_i, a_i), un couple (variable, valeur) et $ListNoSupports(x_i, a_i) = \{(y_1, b_1), ..., (y_m, b_m)\}$, l'ensemble des couples ayant des valeurs non compatibles avec (x_i, a_i). Ces couples sont les conflits de (x_i, a_i). Une solution de Π est noté Sol_Π.

3 SBS : Search Based Substituality

3.1 Substituabilité

Definition 1 *[3] Soit a_i et b_i, deux valeurs de $D(x_i)$. On dit que a_i est* **substituable** *à b_i si et seulement si toute solution contenant b_i demeure une solution de Π si on remplace b_i par a_i.*

Cette notion a été introduite par Freuder [3] pour accélérer la recherche de solution pour les CSPs en liant deux ou plusieurs valeurs d'une même variable. Par exemple, on a deux couples distincts $(x_i, 1)$ *et* $(x_i, 2)$, la valeur 1 est substituable à la valeur 2 si et seulement si pour toute solution Sol_Π avec $x_i = 2$, il existe une solution Sol'_Π avec $x_i = 1$. SBS utilise ce principe pour compresser des valeurs dans des solutions génériques qui seront plus tard utilisées pour générer toutes les solutions. Pour cela, l'algorithme SBS va provoquer l'apparition des valeurs substituables. Dans les sections suivantes, nous allons détailler l'approche classique d'énumération de solutions ainsi que l'approche SBS.

3.2 Énumération classique de solutions

L'énumération des solutions sur un arbre binaire se fait de la façon suivante. On choisit une variable x_i, puis on choisit une valeur a_i ensuite on instancie $x_i = a_i$ et on propage les conséquences de cette instanciation. Lors de la présence d'un échec (une variable quelconque du CSP Π ayant un domaine vide), on fait un retour arrière (backtrack) et on lance la propagation avec $x_i \neq a_i$. Lorsqu'on prend une décision ($x_i = a_i$ ou $x_i \neq a_i$), on crée un point de choix. Ce dernier est également représenté par un noeud de l'arbre de recherche. Après chaque point de choix, on répète le processus. Lorsque toutes les variables sont instanciées, on a trouvé une solution. L'algorithme ClassicSearch (1) illustre cette approche. On utilise des stratégies de sélection de variables et de valeurs pour sélectionner le couple (variable, valeur) à chaque point de choix. Lorsque toutes les variables sont instanciées, on ajoute la solution trouvée puis on fait un retour arrière pour chercher une nouvelle solution.

Algorithm 1: function CLASSICSEARCH

```
CLASSICSEARCH(X)
REQUIRE : X ← list of variables
if nbAssignments = |X| then
    addSolution(X)
else
    x_i ← selectVariable(X)
    a_i ← selectValue(D(x_i))
    saveState()
    assign(x_i, a_i)
    if propagate(x_i = a_i) then
        ClassicSearch(X)
    unassign(x_i, a_i)
    restoreState()
    saveState()
    if propagate(x_i ≠ a_i) then
        ClassicSearch(X)
    restoreState()
```

3.3 Principe de l'algorithme SBS

Definition 2 *Un* **Tuple Sequence of Interchangeable Elements** *noté TSIE, est un n-tuples $\Delta = (\delta_1, ..., \delta_n)$, où chaque δ_i est un ensemble de valeurs non vide, tel que chaque n-tuple $(v_1, ..., v_n) \in \delta_1, ..., \delta_n$ et $v_n \in D_n$ est soit une solution Sol_Π ou soit dans un état impossible pour le problème Π.*

L'algorithme SBS (2) a une approche différente. Il n'énumère pas explicitement toutes les solutions. Il calcule des solutions génériques à partir desquelles l'ensemble des solutions sera calculé. Á un niveau donné, l'algorithme classique sélectionne une variable puis essaie successivement chaque valeur de cette variable. L'algorithme SBS procède différemment. On identifie le couple (x_i, a_i) qui a le moins de conflits. On instancie d'abord chaque conflit de (x_i, a_i) et on instancie (x_i, a_i). Á ce moment, Les autres valeurs de $D(x_i)$ n'ont pas besoin d'être instanciées avec x_i, car elles ne sont plus en conflit avec les valeurs des $D(x_j)$. En procédant ainsi, on espère réduire l'espace de recherche. Quand on instancie (x_i, a_i), on place les valeurs restantes de $D(x_i)$ dans $TSIE$. Enfin, on instancie (x_i, a_i). Si le nombre de conflits est strictement inférieur au

Algorithm 2: function SBS

```
SBS(X)
REQUIRE : X ← list of variables
if nbAssignments = |X| then
    TSIE-enumeration(TSIE)
else
    (x_i, a_i) ← getVariableValueMinConflicts()
    for each (y, b) ∈ ListNoSupports(x_i, a_i) do
        TSIE[y] ← TSIE[y] ∪ {b}
        saveState()
        assign(y, b)
        if propagate(y = b) then
            SBS(X)
        unassign(y, b)
        restoreState()
        TSIE[y] ← TSIE[y] − {b}
        if not propagate(y ≠ b) then
            return
    TSIE[x_i] ← TSIE[x_i] ∪ D(x_i)
    saveState()
    assign(x_i, a_i)
    if propagate(x_i = a_i) then
        SBS(X)
    unassign(x_i, a_i)
    restoreState()
    TSIE[x_i] ← TSIE[x_i] − D(x_i)
```

Algorithm 3: function TSIE-ENUMERATION

```
TSIE-ENUMERATION(TSIE) : generic solution
S_n ← TSIE[n]
for each i from n − 1 to 1 do
    S_i ← ∅
    for each GCS ∈ S_{i+1} do
        newS ← ∅
        for each a_i ∈ TSIE[i] do
            // Calcul de la nouvelle GCS pour la valeur a_i
            newGCS[i] ← {a_i}
            for each j de i + 1 à n do
                newGCS[j] ←
                compatible(x_i, a_i, x_j) ∩ GCS[j]
            newS ← newS ∪ {newGCS}
        // Concaténation des GCS équivalents
        MERGEEQUIVALENTGCS(newS)
        S_i ← S_i ∪ newS
return S_1
```

nombre de valeurs dans le domaine alors SBS fera moins de choix qu'une méthode classique pour déterminer les solutions courantes contenant les éléments de $D(x_i)$. Une fois que toutes les variables sont instanciées, $TSIE$ devient une solution générique qu'il faudra énumérer en vérifiant la compatibilité des valeurs. Nous allons détailler cette énumération dans la section suivante.

4 Énumération des GCS

Definition 3 *[2] Une* **Global Cut Seed** *notée GCS, est un n-tuples $\Delta = (\delta_1, ..., \delta_n)$, où chaque δ_i est un ensemble de valeurs non vide, tel que chaque n-tuple $(v_1, ..., v_n), v_n \in \delta_1, ..., \delta_n$ et $v_n \in D_n$ est une solution du problème* Π. [1]

D'après la définition, on notera qu'il est facile d'énumérer les solutions dans des GCS. Pour énumérer les solutions, nous construisons des $Global\ Cut\ Seed$ (GCS) définies par Foccaci et Milano [2]. Chaque $TSIE_i$ contient une valeur Sub_i qui est est compatible avec toutes les valeurs des autres $TSIE_j$ avec $j > i$. De plus, cette valeur correspond à la valeur que l'on a explicitement instanciée. On peut donc facilement calculer toutes les solutions contenant cette valeur Sub_i. Pour les autres valeurs, c'est plus délicat, car elles peuvent être incompatibles avec des valeurs des autres $TSIE_j$. Il faut donc tester les compatibilités entre valeurs (3). Nous détaillons le déroulement de cet algorithme avec l'exemple suivant. On fixe X avec 5 variables et $TSIE = [\{a_1, b_1\}, \{a_2, b_2, c_2\}, \{c_3\}, \{d_4\}, \{e_5, f_5, g_5\}]$, la liste des ensembles de valeurs $TSIE_i$ générée par l'algorithme (2). Les contraintes de Π définissent les incompatibilités suivantes : e_5 incompatible avec b_2 et c_2, g_5 incompatible avec b_1 et c_3 incompatible avec b_1. Soit $GCS[i]$, l'ensemble des valeurs de la variable x_i de la GCS. L'algorithme satisfait la propriété suivante : à chaque fin d'étape i, on crée une liste S_i qui possède les ensembles des valeurs compatibles de x_i à x_n. Cette liste S_i regroupe des $Global\ Cut\ Seed$. En effet, à l'initialisation $i = 0$, on crée une $Global\ Cut\ Seed$ avec les valeurs supports de la dernière variable instanciée. Ensuite, à chaque étape i, on crée la liste S_i des $Global\ Cut\ Seed$ à partir des $Global\ Cut\ Seed$ générées à l'étape $i - 1$. Pour cela, on ajoute une valeur compatible si et seulement si après élimination des valeurs non compatibles aucun ensemble n'est vide. Pour calculer toutes les solutions de TSIE, on commence par la dernière étape qui génère la première GCS $[\{e_5, f_5, g_5\}]$, ensuite à chaque étape $i - 1$, on vérifie la compatibilité de chaque valeur de l'ensemble $TSIE_i$ avec les autres valeurs des autres ensembles précédents. Ainsi, à l'étape 4, on ajoute l'ensemble $\{d_4\}$, ce qui nous donne S_4 contenant $[\{d_4\}, \{e_5, f_5, g_5\}]$. On continue de la même façon pour la liste S_3, on a donc $[\{c_3\}, \{d_4\}, \{e_5, f_5, g_5\}]$. Á l'étape 2, on a e_5 qui est non compatible avec b_2 et c_2, ainsi la liste S_2 3 GCS : $[\{a_2\}, \{c_3\}, \{d_4\}, \{e_5, f_5, g_5\}]$, $[\{b_2\}, \{c_3\}, \{d_4\}, \{f_5, g_5\}]$, $[\{c_2\}, \{c_3\}, \{d_4\}, \{f_5, g_5\}]$.

1. Nous avons considéré une partie de la définition décrite par [2], car nous voulons réduire la notion de GCS seulement à une liste de tuples n'ayant que des valeurs compatibles

Enfin, à l'étape 1, on a g_5 qui est non compatible avec b_1 et c_3 qui est non compatible avec b_1. La liste S_1 contient :
$[\{a_1\},\{a_2\},\{c_3\},\{d_4\},\{e_5,f_5,g_5\}]$,
$[\{a_1\},\{b_2\},\{c_3\},\{d_4\},\{f_5,g_5\}]$,
$[\{a_1\},\{c_2\},\{c_3\},\{d_4\},\{f_5,g_5\}]$,
$[\{b_1\},\{b_2\},\{c_3\},\{d_4\},\{f_5,g_5\}]$,
$[\{b_1\},\{c_2\},\{c_3\},\{d_4\},\{f_5\}]$.
La GCS $[\{b_1\},\{a_2\},\{\},\{d_4\},\{e_5,f_5,g_5\}]$ n'est pas valide, car il y a la présence d'un ensemble vide. Après la génération de cinq GCS à l'étape 1, on les énumère afin de générer toutes les solutions émanant de cette solution générique. Ainsi, on aura au final dix solutions :
$[\{a_1\},\{a_2\},\{c_3\},\{d_4\},\{e_5\}]$,
$[\{a_1\},\{a_2\},\{c_3\},\{d_4\},\{f_5\}]$,
$[\{a_1\},\{a_2\},\{c_3\},\{d_4\},\{g_5\}]$,
$[\{a_1\},\{b_2\},\{c_3\},\{d_4\},\{f_5\}]$,
$[\{a_1\},\{b_2\},\{c_3\},\{d_4\},\{g_5\}]$,
$[\{a_1\},\{c_2\},\{c_3\},\{d_4\},\{f_5\}]$,
$[\{a_1\},\{c_2\},\{c_3\},\{d_4\},\{g_5\}]$,
$[\{b_1\},\{b_2\},\{c_3\},\{d_4\},\{f_5\}]$,
$[\{b_1\},\{b_2\},\{c_3\},\{d_4\},\{g_5\}]$,
$[\{b_1\},\{c_2\},\{c_3\},\{d_4\},\{f_5\}]$.
Dans cet exemple, la compression des solutions est d'un facteur 2, car on génère 5 GCS pour 10 solutions.
On remarque que la génération de nouvelles GCS peut amener à avoir des listes de GCS identiques. En effet dans l'exemple présenté, on peut observer que pour la liste S_2, on a deux GCS $[\{b_2\},\{c_3\},\{d_4\},\{f_5,g_5\}]$, $[\{c_2\},\{c_3\},\{d_4\},\{f_5,g_5\}]$ qui ont en commun $[\{c_3\},\{d_4\},\{f_5,g_5\}]$. Pour éviter la redondance des GCS, l'algorithme appelle la fonction MERGEEQUIVALENTGCS($newS$) qui recherche les GCS identiques pour la nouvelle GCS créée. Pour cela, on transforme la GCS en une séquence de valeurs. Ensuite, on fait un tri lexicographique pour déterminer s'il y a des égalités entre les GCS et on concatène les GCS identiques. La complexité de notre algorithme est la suivante. Le coût du tri lexicographique est $O(n\log(n))$ comparaisons. Chacune coûtant la somme des valeurs de la GCS. Dans notre cas, on a d éléments considérés et la taille de la chaine maximale est nd. On aura donc un coûte en $O(d\log(d)nd)$, soit $O(nd^2\log(d))$. Dans notre exemple, on aura donc après concaténation des 2 GCS $[\{b_2\},\{c_3\},\{d_4\},\{f_5,g_5\}]$, $[\{c_2\},\{c_3\},\{d_4\},\{f_5,g_5\}]$, une seule GCS $[\{b_2,c_2\},\{c_3\},\{d_4\},\{f_5,g_5\}]$. On peut essayer de concaténer les GCS de plusieurs façons différentes, mais il faut obligatoirement avoir des solutions uniques sinon l'énumération peut devenir difficile.

5 Résultats expérimentaux

Nous présentons dans cette section une évaluation expérimentale de nos algorithmes. Nous proposons de comparer l'approche SBS et l'approche classique d'énumération de solutions avec des différentes stratégies de sélection de couples (variable, valeur). Ensuite, nous évaluons l'efficacité de la concaténation des GCS. Toutes les expériences sont effectuées sur un solveur ad-hoc avec une machine disposant d'un processeur Intel Core i7 de quatre coeurs cadencés à 2,2 GHz. Chaque expérience est exécutée sur un seul coeur. On note $MinDom$, la stratégie de sélection de variables qui sélectionne la variable x_i ayant la plus petite taille du domaine. $MinConflict$ et $MaxConflict$ sont deux stratégies de sélection de valeurs qui sélectionnent la valeur a_i de $D(x_i)$ ayant respectivement le minimum et le maximum de conflits. Nous avons testé SBS et les différentes stratégies de recherche $MinDom/MinConflict$ et $MinDom/MaxConflict$. D'après les résultats, nous n'avons pas observé de différences significatives dans les temps de résolution entre $MinDom/MinConflict$ et $MinDom/MaxConflict$ sur l'ensemble des instances résolues. Ainsi, nous considérons seulement la comparaison entre SBS et $MinDom/MaxConflict$. La génération des instances se fait de manière aléatoire en fixant un nombre de variables V, un nombre de valeurs par domaine D, un nombre de contraintes en extension C, et une dureté des contraintes T ($tightness$ en anglais) qui définit le nombre de tuples non autorisés pour chaque contrainte C. Nous considérons seulement des instances ayant plus de 1000 solutions. On rappelle que les problèmes faiblement contraints avec une $tightness$ faible présentent beaucoup de solutions alors que ceux fortement contraints donc avec une $tightness$ élevée ont peu de solutions. Dans

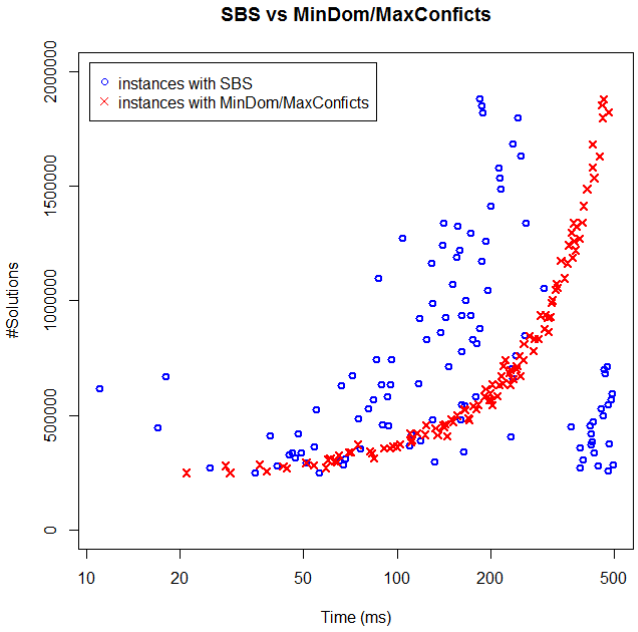

FIGURE 1 – Comparaison des temps de résolution entre SBS et $MinDom/MaxConflicts$

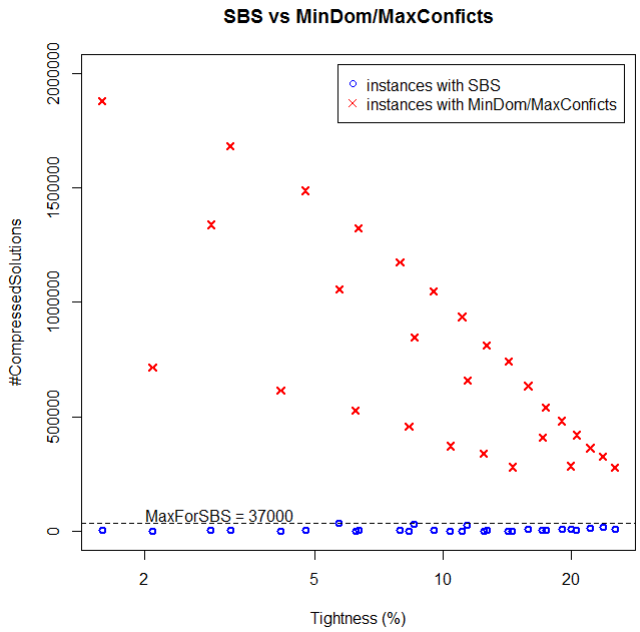

FIGURE 2 – Comparaison des solutions compressées entre SBS et $MinDom/MaxConflicts$

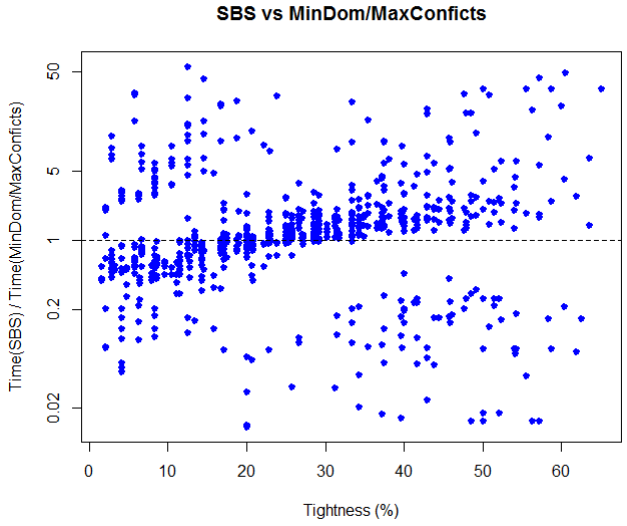

FIGURE 3 – Comparaison des ratios entre SBS et $MinDom/MaxConflicts$

les figures suivantes, chaque instance résolue est représentée par un point. Nous avons vérifié que le nombre de solutions trouvées pour chaque instance est exactement le même entre SBS et $MinDom/MaxConflict$.
La figure 1 compare les temps des deux méthodes pour énumérer toutes les solutions de chaque instance. Ici, SBS utilise la concaténation des GCS dans l'énu-

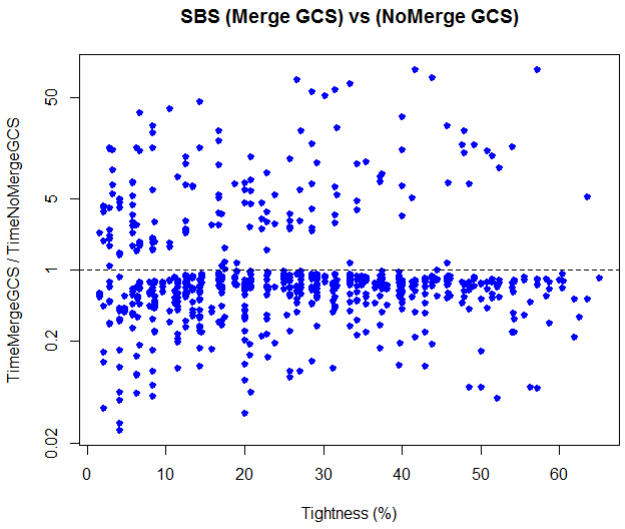

FIGURE 4 – Comparaison des ratios entre SBS avec et sans concaténation des GCS

mération des TSIE. La courbe décrite par la stratégie $MinDom/MaxConflict$ est homogène ainsi plus l'instance possède de solutions et plus le temps est long. Quant aux instances résolues par SBS, elles ont des temps de résolutions variables et sont éparpillées entre le dessous et le dessus de la courbe de $MinDom/MaxConflict$. Ainsi, SBS peut résoudre plus rapidement des instances que $MinDom/MaxConflict$ et plus lentement pour d'autres. Nous allons voir en détail ce qui se passe pour les instances résolues entre SBS et $MinDom/MaxConflict$. La figure 2 compare le nombre de solutions compressées par SBS (solutions génériques) et le nombre de solutions que trouve généralement une stratégie de recherche classique comme $MinDom/MaxConflict$ en fonction de la $tightness$ ici calculée en pourcentage. La compression des solutions avec SBS est impressionnante lorsque la $tightness$ est faible (pour une $tightness$ évaluée inférieure à 25%). En effet, le facteur de compression varie entre 2 et 50 (rapport entre nombre de solutions et le nombre de solutions compressées). On constate qu'au fur et à mesure que celle-ci augmente, la compression de solutions commence à perdre en efficacité. Le facteur de compression est quasiment nul à partir de 50% de $tightness$. Ainsi, déterminer la $tightness$ du problème est très importante afin de pouvoir évaluer à l'avance la stratégie de recherche la plus efficace pour calculer toutes les solutions. Nous avons testé l'efficacité de la concaténation des GCS évoquée dans l'algorithme (3). La figure 3 compare les ratios entre le temps de résolution de SBS et celui de $MinDom/MaxConflict$ en fonction de la $tightness$ ici calculée en pourcentage. SBS utilise la concaténation des GCS dans l'énumération des TSIE. On trace la droite $y = 1$

pour évaluer les ratios sur le graphique. Si une instance est résolue plus rapidement avec SBS, il doit se trouver en dessous de la droite $y = 1$ et inversement. On remarque que pour beaucoup de problèmes faiblement contraints, SBS est beaucoup plus rapide que $MinDom/MaxConflict$ jusqu'à environ 25% de $tightness$. Après ce plafond, $MinDom/MaxConflict$ est plus efficace. C'est prévisible, car comme le montre la figure 2, ce plafond coordonne avec l'écart qui se réduit entre le nombre de solutions compressées par SBS et le nombre de solutions non compressées que trouve $MinDom/MaxConflict$.
Dans la figure 4, nous comparons les ratios entre le temps de résolution de SBS avec concaténation des GCS et celui de SBS sans concaténation des GCS en fonction de la $tightness$ ici calculée en pourcentage. On trace la droite $y = 1$ pour évaluer les ratios sur le graphique, ainsi si une instance est résolue plus rapidement avec SBS avec concaténation des GCS, il doit se trouver en dessous de la droite $y = 1$ et inversement. Avec une $tightness$ faible de l'ordre de moins de 5%, les deux approches sont équivalentes, mais dépassé ce plafond, on observe que pour la plupart des instances, la concaténation des GCS lors de leur énumération permet d'améliorer considérablement le temps de résolution (2 à 5 fois plus rapide).
L'ensemble de ces résultats expérimentaux montre que SBS est très efficace dans le calcul de toutes les solutions des problèmes peu contraints par rapport à une stratégie de recherche classique. Avec des $tightness$ allant de 0% à 25%, on réduit le temps de résolution par 3 en moyenne. La concaténation des GCS améliore sensiblement l'énumération des TSIE avec des gains de temps considérables (2 à 5 fois plus rapide pour la plupart des instances).

6 Conclusion

Nous avons proposé une nouvelle stratégie de recherche basée sur la substituabilité (SBS) pour calculer toutes les solutions d'une instance. D'après les résultats expérimentaux, SBS améliore les problèmes d'un facteur 2 à 5 en temps de résolution et d'un facteur de 2 à 50 en terme de mémoire. Mais, cette approche perd en efficacité quand la $tightness$ augmente au-delà de 25%. Nous avons comparé les différentes stratégies de recherche et nous avons évalué l'efficacité de la concaténation des GCS pour l'énumération des solutions. L'algorithme présenté ne considère que les contraintes binaires en extension, il serait intéressant de généraliser cette idée pour les CSP classiques invoquant notamment des contraintes globales.

Références

[1] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph. *Communications of the ACM*, pages 575–577, 1973.

[2] Focacci and Milano. Global cut framework for removing symmetries. pages 77–92, 2001.

[3] Eugene C. Freuder. Eliminating interchangeable values in csp. pages 227–233, 1991.